Lab 11

Demo Points (50 Marks)

Stories 14 – 24 (5 Marks each)

- JUnit Test (3 Marks)
- LibraryInOut method (2 Marks)

This Week

This week you will add the capability to load and save your library.

The design changes again. Some of the changes are cosmetic. We have moved the Friend class to the right and the Item class to the left. This is just to make the classes fit on the page better. It is wise to try to make your class diagrams as neat as possible because tidying them up gives you a chance to think about what they mean. Also, they are easier to make sense of when they are tidier.

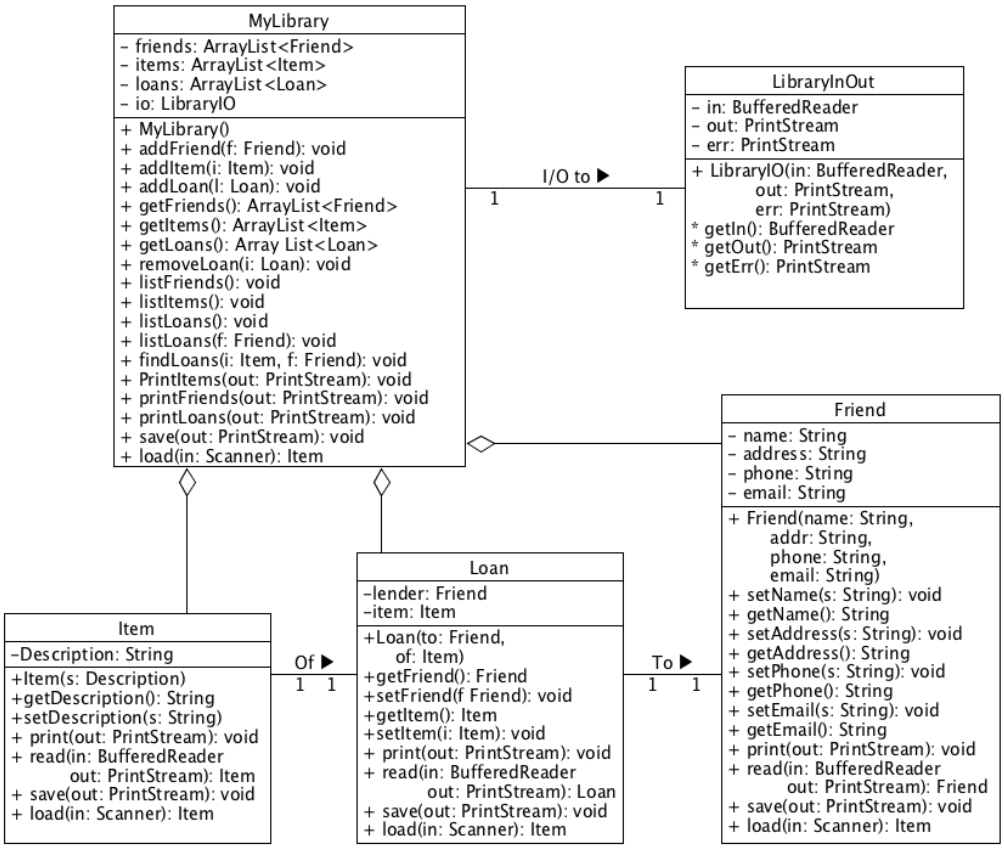

Illustration 1: Refactored Library

Notice that we have added a PrintStream to all of the print() functions. This will let us pass the stream we want to print to into the function. We also pass a BufferedReader and a PrintStream to the read() functions. The intention is that the read() functions will write prompts to the PrintStream and read results from the BufferedReader. For a Command Line Interface (CLI), the PrintStream will be created from the

standard output and the BufferedReader will be created from standard input.

More importantly, we have added load() and save() methods to Item, Loan, Friend, and MyLibrary. These methods will save the contents of the object on which they are called to a file pointed to by the PrintStream and will load the same item from the file pointed to by the Scanner. Scanners are discussed more below. With these two new methods we may now store a library and load it back in.

Unfortunately, this raises new issues when we try to load and save loans. The problem is that we do not want to create a new friend and item when we load a loan, instead we want the loan to contain the Friend and Item that we already have in the list. To do this, we introduce id strings to the Item and Friend class. We will store these ids instead of the object itself when we save the object, and we will find the object in the items and friends lists of MyLibrary. To do this we will need to be able to find the item in the items and friend in the friends. This task will be performed by the findFriend() method which searches the list for an id and returns the Friend it finds in the friends list and the findItem() method, which does the same for the items list.

Ids need only differ from all other ids. Therefore, we can compute these as we create the new object. For example, here is the new constructor for an Item that creates a unique id. Notice that we use a static count and increment it each time we create a new id so that all of the Item ids are unique.

```
public Item(String description) {
    this.id = "i" + itemCounter++;
    this.description = description;
}
```

Illustration 2: New constructor for Item that adds an id

Saving and Loading Item and Friend

Saving an Item or a Friend is relatively easy: one simply formats the output to the stream that is passed in. It is different from printing the Item or Friend because the program will load it by reading the items from the file into which we saved it. We do not care as much if a human can understand it, but we care a lot that the program can parse it easily. To make it easy to parse, we will store one item per line. Each field in the item will be separated from the rest by a tab character. The tab character will be relatively easy for a human to see if they print the line, and it is unlikely that the fields will contain tabs themselves. For example, an item with the description field “Harry Potter and the Sorcerer's Stone” will have the form “Item<tab>Harry Potter and the Sorcerer's Stone<newline>”. Therefore, the save function will just save all of the fields in an Item or a Friend on a line that starts with either “Item” or “Friend” separated by tabs. For example, the test function for save() for Items will look like this:

```
@Test
public void testSaveItem() {
    ByteArrayOutputStream stuffPrinted = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(stuffPrinted);

    item = new Item("I1", "test");
    item.save(out);
    assertEquals("Item\tI1\ttest\n", stuffPrinted.toString());
}
```

Illustration 3: The test function for item.save()

Loading will differ from reading more than saving differs from printing. To load an item, one needs to parse the line using tabs as separators. Fortunately, the Scanner class provides exactly the kind of functionality we need. The Scanner class makes reading complex data from a text file much easier than it is in C.

The Scanner class's constructor function takes either a string or a file. A scanner constructed with a string will read the characters in that string as if they came from a file. It makes it easy to create a fake for the load function. For example, the test function for load() for Items will look like Illustration 4. notice that we add “test three words” as the description to make sure that it will deal with white space appropriately.

```
    @Test
    public void testLoadItem() {
        Scanner scanner = new Scanner("Item\tI1\ttest three words\n");

        assertEquals("none", item.getId());
        assertEquals("thingy", item.getDescription());
        assertEquals("Item", scanner.next());
        item = Item.load(scanner);
        assertEquals("I1", item.getId());
        assertEquals("test three words", item.getDescription());
        scanner.close();
    }
```

Illustration 4: The test function for item.load()

Scanner objects also let you set the whitespace character. In this case, we want to set the white space character to “\t”. Now, the scanner.next() call will return the sequence of characters up to the next tab. To do this, we set the delimiter using the useDelimiter() method on the Scanner object, passing in the parameter “\t”. This has the effect of using only tab as the delimiter, not any whitespace.

```
    public static Item load(Scanner scanner) {
        scanner.useDelimiter("\t");
        String id = scanner.next().trim();
        String description = scanner.next().trim();
        return new Item(id, description);
    }
```

Illustration 5: The load() method for Item

The print() and read() functions for a Friend will be a little more difficult because you need to read multiple fields to make a Friend.

Saving and Loading Loan

Saving and loading loans is more difficult that reading and printing loans, because we want to save them in such a way that a Loan associates an Item and a Friend. We need to load the same associate. If

we simply read in a Friend and an Item and associate those two, it will not refer to the Friend and Item that were loaded in earlier.

Instead of loading and saving contents of the Item and Friend, we need to save a pointer to the Item and Friend that are already in the list of items and Friends. For this we will use the ID. To save the Loan, we save the id for the Friend and the id for the Item. To add the ID, we will need to refactor Friend, Item, and Loan. We will add a computed id to use in production as shown in Illustration 6.

```
public Item(String description) {
    this.id = "i" + itemCounter++;
    this.description = description;
}
```

Illustration 6: Computing item id in constructor

We will also need to be able to specify an ID for testing, so we also add a new constructor that takes a string ID.

```
public Item(String id, String description) {
    this.id = id;
    this.description = description;
}
```

Illustration 7: Item constructor with ID parameter

We can now write the save function for loans. First, we write the test. As you can see, the function will print out, the word “Loan”, followed by the ID for the loan, then the ID for the friend, concluded by the id for the item. Each of these is separated by a tab.

```
@Test
public void testSaveLoan() {
    ByteArrayOutputStream stuffPrinted = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(stuffPrinted);

    createLoan();
    loan.save(out);
    assertEquals("Loan\tl1\tf1\ti1\n", stuffPrinted.toString());
}
```

Illustration 8: Test for saving a loan

Saving the loan is simple, reading it back in is more complicated. We need to translate the id of the item into the item itself. Therefore, to read the loan back in we need the library we are reading that is already populated with the friends and items that have been lent. To do this, we will need to pass the library in to the function that will load the loan, and find the friend and item associated in the loan.

We will need a find function for items and a find function for friends. In our design, these functions are in the MyLibrary class: findItem(String Itemid) and findFriend(String Friendid). The test function for findItem is shown in Illustration 9. Notice that I have extracted a method to initialize a library. This method will be useful when we develop other load functions.

```
private MyLibrary initializeLibrary(MyLibrary ml) {
    ml = new MyLibrary();
    ml.addItem(new Item("i1", "test 1"));
    ml.addItem(new Item("i2", "test 2"));
    ml.addItem(new Item("i3", "test 3"));
    ml.addFriend(new Friend("f1", "Alice", "Abc", "123", "alice@x.com"));
    ml.addFriend(new Friend("f2", "Bob", "Def", "456", "bob@y.com"));
    ml.addFriend(new Friend("f3", "Cam", "Ghi", "789", "cam@z.com"));
    ml.addLoan(new Loan("l1", ml.getFriend(0), ml.getItem(2)));
    ml.addLoan(new Loan("l2", ml.getFriend(1), ml.getItem(0)));
    return ml;
}

@Test
public void testFindItem() {
    ml  = new MyLibrary();
    ml = initializeLibrary(ml);

    assertEquals(ml.getItem(0), ml.findItem("i1"));
    assertEquals(ml.getItem(1), ml.findItem("i2"));
    assertEquals(ml.getItem(2), ml.findItem("i3"));
    assertNotEquals(ml.getItem(2), ml.findItem("i1"));
    assertNotEquals(ml.getItem(1), ml.findItem("i3"));
    assertNotEquals(ml.getItem(0), ml.findItem("i3"));
}
```

Illustration 9: Test method for findItem()

Once we can find an item from a library, we can now read a representation of a loan that associates IDs and turn it into a loan that associates objects. Notice that I check that the objects were found and print error messages if they are not. It wold be better to throw an exception because by returning null, we assume that all calling methods will test the result of this function. This requires more knowledge from the calling methods than we should expect: remember, the calling methods may be written by someone in the distant future. To make sure I remember to change it to throw an exception, I add a TODO tag to the function.

```
public static Loan load(Scanner scanner, MyLibrary ml) {

    String loanId = scanner.next().trim();
    Friend f = ml.findFriend(scanner.next().trim());
    Item i = ml.findItem(scanner.next().trim());

    //TODO: throw an exception rather than returning null
    if (f == null && i == null) {
        System.err.println("Loan.load(): friend and item are null");
    } else if (f == null) {
        System.err.println("Loan.load(): friend is null");
    } else if (i == null) {
        System.err.println("Loan.load(): item is null");
    } else {
        return new Loan(loanId, f, i);
    }
    return null;
}
```

Illustration 10: load() method for Loan

Eclipse keeps track of all of the comments that start with TODO in a list in the Tasks window. My Tasks window is shows in Illustration 11. If you click on the task in the window, it will open the file that contains that tag and move the cursor to it. It is a useful way to send notes to your future self.

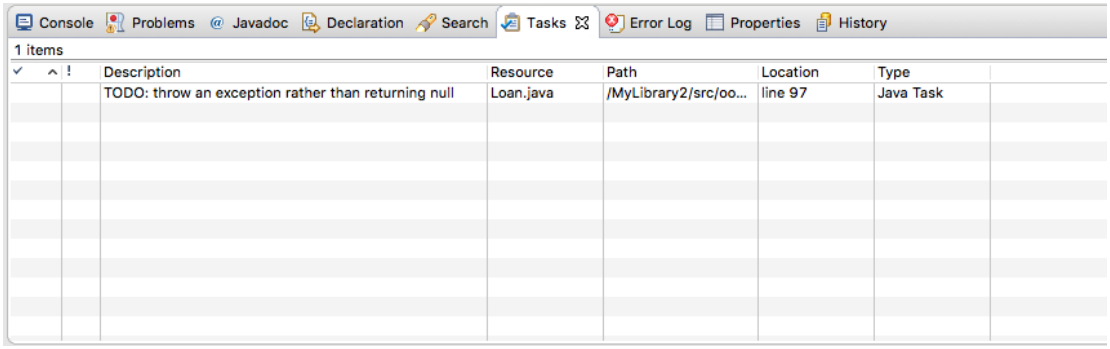

Illustration 11: Task window

Saving and Loading MyLibrary

Once you and store and load an Item, a Friend, and a Loan, it is relatively easy to load and store the items list, the friends list, and the loans list. Writing the lists is most easily accomplished by using the for idiom for lists. For example, this idiom for applying doing things to all of the items in a library is: `for (Item i: items)`. The Item, i, will be given the value of each Item in items in each iteration of the loop.

When loading the file, each line will start with the type of the object, so you can use a case statement to determining what type to add and which list to add it to.

Next Week's Demo Points (100 marks)

Stories 31-44 (5 Marks Each)

- JUnit test (2 Marks)
- Implementation (3 Marks)

Story 45 (35 marks)

- JUnit tests (15 marks)
- Implementation (20 marks)